Extending an Open-source Real-time Operating System with Hierarchical Scheduling*

Mike Holenderski, Wim Cools, Reinder J. Bril, Johan J. Lukkien
Eindhoven University of Technology,
Den Dolech 2, 5600 AZ Eindhoven, The Netherlands

October 11, 2010

Abstract

Hierarchical scheduling frameworks (HSFs) have been devised to support the integration of independently developed and analyzed subsystems. This paper presents an efficient, modular and extendible design for enhancing a real-time operating system with periodic tasks, two-level fixed-priority HSF based on the idling periodic and deferrable servers, and virtual timers. It relies on Relative Timed Event Queues (RELTEQ), a timed event management component targeted at embedded operating systems, which supports long event interarrival time, long lifetime of the event queue, no drift and low overhead. It minimizes the overhead in terms of processor and memory usage, limits the interference of inactive servers on system level, and minimizes the necessary modifications of the underlying operating system. The proposed design was implemented and evaluated within the μC/OS-II real-time operating system used by our industrial and academic partners.

*This work has been supported in part by Information Technology for European Advancement (ITEA2), via the Content Aware Networked Systems Towards Advanced and Tailored Assistance (CANTATA) project.

1 Introduction

A real-time system is usually comprised of a real-time application running on top of a real-time operating system. One such operating system is μC/OS-II [13]. It is maintained and supported by Micrium, and is applied in many application domains, e.g. avionics, automotive, medical and consumer electronics. Our choice of μC/OS-II is in line with our industrial and academic partners at VDG Security and Mälardalen University.

In [9] we have presented an application of our industrial partner in the surveillance domain. We pointed out how a combination of periodic tasks and processor reservations based on deferrable servers, amongst others, can be used to better exploit the available network bandwidth. As a next step we have set out on extending μC/OS-II with the proposed real-time primitives.

1.1 Problem description

We require an interface and a corresponding generic implementation of the following functionalities:

1. Periodic task activation.
2. Hierarchical scheduling support, i.e., the creation of budgets with parameters for subsystem scheduling. Hierarchical scheduling requires means for temporal isolation and prevention of undesirable interference of subsystems, including event expiration for periodic tasks of subsystems which do not have the processor at the moment.
3. Task management within subsystems. As part of this, we need the concept of a timer *relative to a server's budget*, which we call a *virtual timer* as opposed to the more common *global timers* which are relative to a fixed point in time.

A subsystem can be regarded as a combination of a set of tasks, a local scheduler and a budget. A server defines the replenishment and depletion policy for a budget. In this paper we use the accepted approach of fixed-priority periodic servers [22, 4] to map budgets to subsystems and use 'budget' and 'server' interchangeably. In order to implement periodic tasks, we require timers to represent their arrival time. For servers, we also need timed events representing the replenishment and depletion of a budget. Vital, and a starting point for our implementation, is therefore the assumption that an event can be delivered at a certain time. This simple timer support is assumed available through the Real-Time Operating System (RTOS). Some RTOSs provide much more functionality (for which our work then provides an efficient realization) but other systems provide just that. For example, μC/OS-II, which we use as a research and demonstration vehicle, provides just a 'Delay' primitive. It has no support for periodic tasks, servers, or HSF, which are required for implementing the system described in [9]. Moreover, every delay event is represented with a separate counter, which is decremented upon every tick, hence the tick handler complexity is linear in the number of events.

As a result, the emphasis of our work lies with the management of timed events.

These requirements should be met by a modular and extensible design, with low performance overhead and minimal modifications to the underlying RTOS. It should exhibit predictable overhead, while remaining efficient to support resource-constrained embedded systems.

1.2 Contributions

We present a design for the support of periodic tasks, the idling periodic and HSF, including the idling periodic and deferrable servers, and virtual timers. This work is based on our earlier work on Relative Timed Event Queues (RELTEQ) [10][1], an efficient timed event management system targeted at embedded systems. We show how its versatile primitives can be leveraged to satisfy our requirements.

The server extension of RELTEQ provides access to virtual timers and provides a mechanism for tasks to monitor their server's remaining budget. The HSF design provides temporal isolation and supports independent development of subsystems by separating the global and local scheduling, and allowing each server to define a dedicated scheduler. Furthermore, the design addresses the system overheads inherent to an HSF implementation and avoids recalculating the expiration of local events, such as budget depletion and task deadlines, upon every server switch. It also prevents undesirable interference between subsystems, by limiting the interference of inactive servers on system level.

[1] The source code of RELTEQ with the described extensions is available at `http://www.win.tue.nl/~mholende/ucos/`

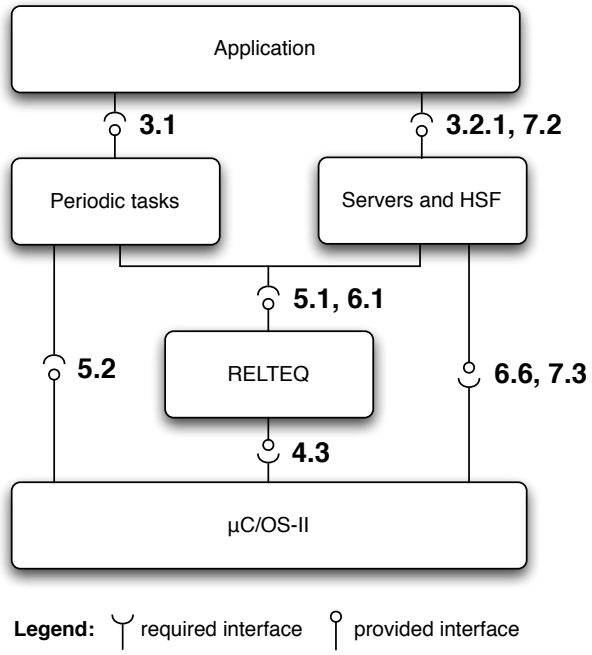


Figure 1: Interfaces between the underlying μC/OS-II and the extensions presented in this paper. The numbers indicate the sections describing the provided interfaces and their implementation.

The design is modular with the interfaces between the different components illustrated in Figure 1. The support for periodic tasks and HSF can be enabled and disabled during compilation.

We show how this design is implemented within μC/OS-II, and provide simulation results demonstrating low overheads of the implementation.

1.3 Outline

We start by discussing the related work in Section 2, followed by a description of the system model and the requirements for the proposed design in Section 3. In Section 4 we summarize the RELTEQ approach to multiplexing timed events on a single hardware timer. We subsequently show how it can be leveraged to support periodic tasks in Section 5, fixed-priority servers in Section 6, and hierarchical scheduling in Section 7. We evaluate the implementation in Section 8 and conclude in Section 9.

2 Related work

In this section we summarize existing work related to hierarchical scheduling and timer management.

2.1 Hierarchical scheduling

[15] introduce the notion of processor reservations, aiming at providing temporal isolation for individual subsystems comprising a real-time system. [19] identify four mechanisms which are required to implement such reservations: *admission control*, *scheduling*, *monitoring* and *enforcement*. [20] present the implementation and analysis of a HSF based on deferrable and sporadic servers using a hierarchical deadline monotonic scheduler.

[12] propose a two-level HSF called the SPIRIT uKernel, which provides a separation between subsystems by using partitions. Each partition executes a subsystem, and uses the Fixed-Priority Scheduling (FPS) policy as a local scheduler to schedule the subsystem's tasks. An offline schedule is used to schedule the partitions on a global level.

[21] introduce the periodic resource model, allowing the integration of independently analyzed subsystems in compositional hard real-time systems. Their resource is specified by a pair (Π_i, Θ_i), where Π_i is its replenishment period and Θ_i is its capacity. They also describe the schedulability analysis for a HSF based on the periodic

resource model under the Earliest Deadline First and Rate Monotonic scheduling algorithms. While the periodic-idling server [4] conforms to the periodic resource model, the deferrable [22] and polling [14] servers do not.

[1] present an implementation of a HSF based on the periodic resource model in the VxWorks operating system. They keep track of budget depletion by using separate event queues for each server in the HSF by means of absolute times. Whenever a server is activated (or switched in), an event indicating the depletion of the budget, i.e. the current time plus the remaining budget, is added to the server event queue. On preemption of a server, the remaining budget is updated according to the time passed since the last server release and the budget depletion event is removed from the server event queue. When the server's budget depletion event expires, the server is removed from the server ready queue, i.e. it will not be rescheduled until the replenishment of its budget.

[16], describe the design and implementation of Linux/RK, an implementation of a resource kernel (Portable RK) within the Linux kernel. They minimize the modifications to the Linux kernel by introducing a small number of call back hooks for identifying context switches, with the remainder of the implementation residing in an independent kernel module. Linux/RK introduces the notion of a resource set, which is a set of processor reservations. Once a resource set is created, one or more processes can be attached to it to share its reservations. Although reservations are periodic, periodic tasks inside reservations are not supported. The system employs a replenishment timer for each processor reservation, and a global enforcement timer which expires when the currently running reservation runs out of budget. Whenever a reservation is switched in the enforcement timer is set to its remaining budget. Whenever a reservation is switched out, the enforcement timer is cancelled, and the remaining budget is recalculated.

AQuoSA [18] also provides the Linux kernel with EDF scheduling and various well-known resource reservation mechanisms, including the constant bandwidth server. Processor reservations are provided as servers, where a server can contain one or more tasks. Periodic tasks are supported by providing an API to sleep until the next period. Similar to [16] it requires a kernel patch to provide for scheduling hooks and updates the remaining budget and the enforcement timers upon every server switch.

[8] present an implementation of the Earliest Deadline First (EDF) and constant bandwidth servers for the Linux kernel, with support for multicore platforms. It is implemented directly into the Linux kernel. Each task is assigned a period (equal to its relative deadline) and a budget. When a task exceeds its budget, it is stopped until its next period expires and its budget is replenished. This provides temporal protection, as the task behaves like a hard reservation. Each task is assigned a timer, which is activated whenever a task is switched in, by recalculating the deadline event for the task.

[6] describe Nano-RK, a reservation-based RTOS targeted for use in resource-constrained wireless sensor networks. It supports fixed-priority preemptive multitasking, as well as resource reservations for processor, network, sensor and energy. Only one task can be assigned to each processor reservation. Nano-RK also provides explicit support for periodic tasks, where a task can wait for its next period. Each task contains a timestamp for its next period, next replenishment and remaining budget. A one-shot timer drives the timer ISR, which (i) loops through all tasks, to update their timestamps and handle the expired events, and (ii) sets the one-shot timer to the next wakeup time.

Unlike the work presented in [1], which implements a HSF on top of a commercial operating system, and in [16, 8, 18], which implement reservations within Linux, our design for HSF is integrated within a RTOS targeted at embedded systems. [12] describe a micro-kernel with a two-level HSF and time-triggered scheduling on the global level. Similar to [18], our work supports various two-level fixed-priority hierarchical processor scheduling mechanisms, including the deferrable [22] and periodic idling [4] servers.

Our design aims at efficiency, in terms of memory and processor overheads, while minimizing the modifications of the underlying RTOS. Unlike [16, 1, 18] it avoids recalculating the expiration of local server events, such as budget depletion or deadline events, upon every server switch. It also limits the interference of inactive servers on system level by deferring the handling of their local events until they are switched in. While [1] present an approach for limiting interference of periodic idling servers, to the best of our knowledge, our work is the first to also cover deferrable servers.

2.2 Timer management

In [16, 18] each timer consists of a 64-bit absolute timestamp and a 32-bit overflow counter. The timers are stored in a sorted linked list. A timer Interrupt Service Routine (ISR) checks for any expiring timers, and performs the actual enforcement, replenishment, and priority adjustments. In [16] the timer ISR is driven by a one-shot high resolution timer which is programmed directly. [18] use the Linux timer interface, and therefore their temporal granularity and latency depend on the underlying Linux kernel.

The [6] implementation is based on the POSIX time structure timeval, with two 32-bit numbers to represent seconds/nanoseconds. The authors assume the timestamp value is large enough that it will practically not overflow.

[3] present the Implicit Circular Timers Overflow Handler (ICTOH), which is an efficient time representation of absolute deadlines in a circular time model. It requires managing the overflow at every time comparison and is limited to timing constraints which do not exceed 2^{n-1}, where n is the number of bits of the time representation. [2] present an implementation of EDF scheduler based on ICTOH for the ERIKA Enterprise kernel [7] and focus on minimizing the tick handler overhead.

In [10] we introduced Relative Timed Event Queues (RELTEQ), which is a timed event management component targeted at embedded operating systems. It supports long event interarrival time, long lifetime of the event queue, no drift and low overhead. In this paper we present the design of an HSF based on RELTEQ, comprising deferrable and idling periodic servers, global and virtual timers. Similar to [5, 12], our RELTEQ implementation is tick based, driven by a periodic hardware timer.

3 System model

In this paper we assume a system is composed of independently developed and analyzed subsystems. A subsystem consists of a set of tasks, which implement the desired application, a local scheduler, and a server. There is a one-to-one mapping between subsystems and servers.

3.1 Tasks

We consider a set Γ of periodic tasks, where each task $\tau_i \in \Gamma$ is specified by a tuple (i, ϕ_i, T_i, C_i), where i is the priority (smaller i means higher priority), ϕ_i is the task's (absolute) offset, T_i is the interarrival time between two consecutive jobs and C_i is its worst-case execution time. Tasks are preemptive and independent.

This task concept is an abstraction of a program text. There are roughly three approaches to periodic tasks, depending on the primitives the operating system provides. Figure 2 illustrates the possible implementations of periodic tasks, where function $f_i()$ represents the body of a task τ_i (the actual work).

```
Task τ_i :
k := 0;
now := GetTime();
DelayFor(φ_i − now);
while true do
   f_i();
   k := k + 1;
   now := GetTime();
   DelayFor(φ_i + k∗T_i − now);
end while
```

(a)

Registration (e.g. in main program):

```
TaskMakePeriodic(τ_i, φ_i, T_i);

Task τ_i :
while true do
   f_i();
   TaskWaitPeriod();
end while
```

(b)

Registration:

```
RegisterPeriodic(f_i(), φ_i, T_i);
```

(c)

Figure 2: Possible implementations of a periodic task.

In the first case, the periodic behavior is programmed explicitly while in the second case this periodicity is implicit. The first syntax is typical for a system without support for periodicity, like μC/OS-II which we study as our running example. It provides two methods for managing time: $GetTime()$ which returns the current time, and $DelayFor(t)$ which delays the execution of the current task for t time units relative to the time when the method was called. As an important downside, the approach in Figure 2.a may give rise to jitter, when the task is preempted between $now := GetTime()$ and $DelayFor()$.

In order to go from Figure 2.a to 2.c we extract the periodic timer management from the task in two functions: a registration of the task as periodic and a synchronization with the timer system. A straightforward implementation of *TaskWaitPeriod()* is a suspension on a semaphore. Going from interface in Figure 2.b to 2.c is now a simple implementation issue.

Note that the task structure described in Figure 2.b guarantees that a job will not start before the previous job has completed, and therefore makes sure that two jobs of the same task will not overlap if the first job's response time exceeds the task's period.

3.2 Servers

We consider the idling periodic and deferrable server, which conform to the periodic resource model [21] and therefore allow for independent development and analysis of subsystems. We consider a set of servers Σ, where each server $\sigma_i \in \Sigma$ is specified by a tuple (i, Π_i, Θ_i), where i is the priority (smaller i means higher priority), Π_i is its replenishment period and Θ_i is its capacity. During runtime, its available budget β_i may vary. Every Π_i time units β_i is replenished to Θ_i. When a server is running, every time unit its available budget β_i is decremented by one.

The mapping of tasks to servers is given by $\gamma(\sigma_i) \subseteq \Gamma$ which defines the set of tasks mapped to server σ_i. We assume that each task is mapped to exactly one server, i.e. $\bigcup_{\sigma_i \in \Sigma} \gamma(\sigma_i) = \Gamma$ and $\bigcap_{\sigma_i \in \Sigma} \gamma(\sigma_i) = \emptyset$. A task $\tau_j \in \gamma(\sigma_i)$ which is mapped to server σ_i can execute only when $\beta_i > 0$.

3.2.1 Server interface

A server σ_i is created using *ServerCreate*$(\Pi_i, \Theta_i, kind)$, where $kind$ specifies whether the server is *idling periodic* or *deferrable*. A task τ_i is mapped to server σ_i using *ServerAddTask(σ_i, τ_i)*.

3.2.2 Deferrable server

The deferrable server [22] is bandwidth preserving. This means that when a server is switched out because none of its tasks are ready, it will preserve its budget to handle tasks which may become ready later. A deferrable server can be in one of the states shown in Figure 3. A server in the *running* state is said to be *active*, and in either *ready*, *waiting* or *depleted* state is said to be *inactive*. A change from inactive to active or vice-versa is accompanied by the server being *switched in* or *switched out*, respectively.

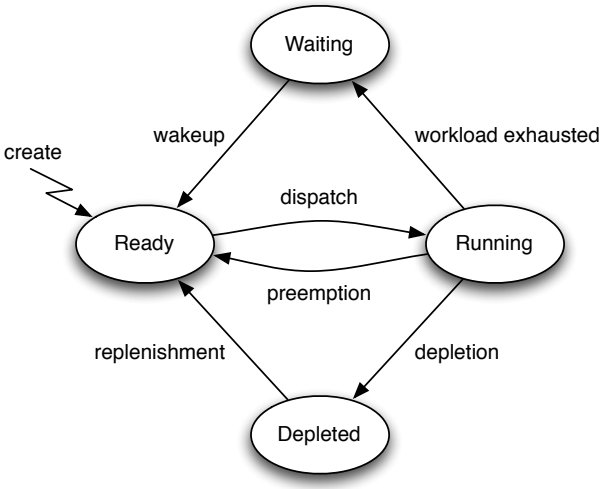


Figure 3: State transition diagram for the deferrable server. The replenishment transitions from the Ready, Running and Waiting states pointing to the same state are not shown.

A server σ_i is created in the ready state, with $\beta_i = \Theta_i$. When it is dispatched by the scheduler it moves to running state. A running server may become inactive for one of three reasons:

- It may be preempted by a higher priority server, upon which it preserves its budget and moves to the ready state.
- It may have available budget $\beta_i > 0$, but none of its tasks in $\gamma(\sigma_i)$ may be ready to run, upon which it preserves its budget and moves to the waiting state.

- Its budget may become depleted, upon which it moves to the depleted state.

When a depleted server is replenished it moves to the ready state and becomes eligible to run. A waiting server may be woken up by a newly arrived periodic task or a delay event. While a server is in the ready or running state it may be replenished, however, since it does not impact the server's state, we have omitted these transitions in the figure.

3.2.3 Idling periodic server

When the idling periodic server [4] is replenished and none of its tasks are ready, then it idles its budget away until either a proper task arrives or the budget depletes. An idling periodic server follows the state transition diagram in Figure 3, however, due to its idling nature it will never reach the waiting state.

3.2.4 Polling server

The polling server [14] is not bandwidth preserving. Therefore, when it is replenished and none of its tasks are ready, then its budget is immediately depleted, and any depletion event already in its virtual server queue is deleted. The same happens when it is switched out.

A polling server can be in one of three states, shown in Figure 4. The difference between a polling and a

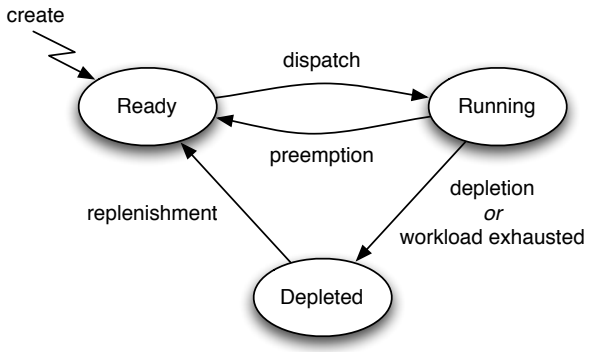


Figure 4: State transition diagram for the polling server.

deferrable server lies in what happens when the workload of a running server is exhausted. Rather than moving to the waiting state (and thus preserving its budget), the polling server discards any remaining budget and moves to the depleted state.

3.3 Hierarchical scheduling

In two-level hierarchical scheduling one can identify a global scheduler which is responsible for selecting a subsystem. The subsystem is then free to use any local scheduler to select a task to run.

In order to facilitate the reuse of existing subsystems when integrating them to form larger systems, the platform should support (at least) fixed-priority preemptive scheduling at the local level within subsystems (since it is a de-facto standard in the industry). To give the system designer the most freedom it should support arbitrary schedulers at the global level. In this paper we will focus on a fixed-priority scheduler on both local and global level.

3.4 Timed events

The platform needs to support at least the following timed events: task delay, arrival of a periodic task, server replenishment and server depletion.

Events local to server σ_i, such as the arrival of periodic tasks $\tau_j \in \gamma(\sigma_i)$, should not interfere with other servers, unless they wake a server, i.e. the time required to handle them should be accounted to σ_i, rather than the currently running server. In particular, handling the events local to inactive servers should not interfere with the currently active server and should be deferred until the corresponding server is switched in.

4 RELTEQ

To implement the desired extensions in μC/OS-II we needed a general mechanism for different kinds of timed events, exhibiting low runtime overheads. This mechanism should be expressive enough to easily implement higher level primitives, such as periodic tasks, fixed-priority servers and two-level fixed-priority scheduling.

4.1 RELTEQ time model

RELTEQ stores the arrival times of future events relative to each other, by expressing their time *relative to their previous event*. The arrival time of the head event is relative to the current time[2], as shown in Figure 5.

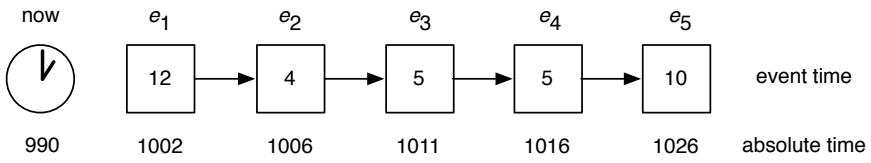


Figure 5: Example of the RELTEQ event queue.

Unbounded interarrival time between events

For an n-bit time representation, the maximum interval between two consecutive events in the queue is $2^n - 1$ time units[3]. RELTEQ improves this interval even further and allows for an *arbitrarily long interval* between *any* two events by inserting "dummy" events, as shown in Figure 6.

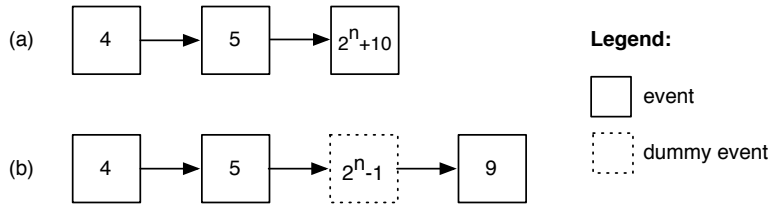


Figure 6: Example of (a) an overflowing relative event time (b) RELTEQ inserting a dummy event with time $2^n - 1$ to handle the overflow.

If t represents the event time of the last event in the queue, then an event e_i with a time larger than $2^n - 1$ relative to t can be inserted by first inserting dummy events with time $2^n - 1$ at the end of the queue until the remaining relative time of e_i is smaller or equal to $2^n - 1$.

In general, dummy events act as placeholders in queues and can be assigned any time in the interval $[0, 2^n - 1]$.

4.2 RELTEQ data structures

A RELTEQ *event* is specified by the tuple *(kind, time, data)*. The *kind* field identifies the event kind, e.g. a delay or the arrival of a periodic task. *time* is the event time. *data* points to additional data that may be required to handle the event and depends on the event kind. For example, a delay event will point to the task which is to be resumed after the delay event expires. Decrementing an event means decrementing its event time and incrementing an event means incrementing its event time. We will use a dot notation to represent individual fields in the data structures, e.g. $e_i.time$ is the event time of event e_i.

A RELTEQ *queue* is a list of RELTEQ events. *Head(q_i)* represents the head event in queue q_i.

[2]Later in this paper we will use RELTEQ queues as an underlying data structure for different purposes. We will relax the queue definition: all event times will be expressed relative to their previous event, but the head event will not necessarily be relative to "now".

[3]With n bits we can represent 2^n distinct numbers. Since we start at 0, the largest one is $2^n - 1$.

4.3 RELTEQ tick handler

While RELTEQ is not restricted to any specific hardware timer, in this paper we assume a periodic timer which invokes the *tick handler*, outlined in Figure 7.

The tick handler is responsible for managing the *system queue*, which is a RELTEQ queue keeping track of all the timed events in the system. At every tick of the periodic timer the time of the head event in the queue is decremented. When the time of the head event is 0, then the events with time equal to 0 are popped from the queue and handled.

The scheduler is called at the end of the tick handler, but only in case an event was handled. If no event was handled the currently running task is resumed straightway.

The behavior of a RELTEQ tick handler is summarized in Figure 7.

Head(system).time := *Head(system).time* - 1;
if $Head(system).time = 0$ **then**
  while $Head(system).time = 0$ **do**
    $HandleEvent(Head(system))$;
    $PopEvent(system)$;
  end while
  $Schedule()$;
end if

Figure 7: Pseudocode for the RELTEQ tick handler.

How an event is handled by *HandleEvent()* depends on its kind. E.g. a *delay event* will resume the delayed task. In general, the event handler will often use the basic RELTEQ primitives, as described in the following sections.

Since we replace the standard μC/OS-II tick handler, it is indicted in Figure 1 as an interface provided by RELTEQ and required by μC/OS-II.

Note that the tick granularity dictates the granularity of any timed events driven by the tick handler: e.g. a server's budget can be depleted only upon a tick. High resolution one-shot timers (e.g. High Precision Event Timer) provide a fine grained alternative to periodic ticks. In case these are present, RELTEQ can easily take advantage of the fine time granularity by setting the timer to the expiration of the earliest event among the active queues. The tick based approach was chosen due to lack of hardware support for high resolution one-shot timers on our example platform. In case such a one-shot timer is available, our RELTEQ based approach can be easily modified to take advantage of it.

5 Periodic tasks

In Section 3.1 we outlined the implementation of a periodic task, where at the end of each job the task acquires a semaphore inside *TaskWaitPeriod()*. In order to support periodic tasks we need to implement a timer which expires periodically and triggers the task waiting inside the *TaskWaitPeriod()* call. In this section we use the periodic task requirements for a periodic timer to introduce the basic RELTEQ primitives, followed by an outline of the periodic task implementation using these primitives.

5.1 Basic RELTEQ primitives

Three operations can be performed on an event queue: a new event can be inserted, the head event can be popped, and an arbitrary event in the queue can be deleted.

NewEvent(k, t, p) Creates and returns a new event e_i with $e_i.kind = k$, $e_i.time = t$, and $e_i.data = p$.

InsertEvent(q_i, e_j) When a new event e_j with absolute time t_j is inserted into the event queue q_i, the queue is traversed accumulating the relative times of the events until a later event e_k is found, with absolute time $t_k \geq t_j$. When such an event is found, then (i) e_j is inserted before e_k, (ii) its time $e_j.time$ is set relative to the previous event, and (iii) the arrival time of e_k is set relative to e_j (i.e. $t_k - t_j$). If no later event was found, then e_j is appended at the end of the queue, and its time is set relative to the previous event.

PopEvent(q_i) When an event is popped from a queue it is simply removed from the head of the queue q_i. It is not handled at this point.

DeleteEvent(q_i, e_j) Since all events in a queue are stored relative to each other, the time $e_j.time$ of any event $e_j \in q_i$ is critical for the integrity of the events later in the queue. Therefore, before an event e_j is removed from q_i, its event time $e_j.time$ is added to the following event in q_i.

Note that the addition could overflow. In such case, instead of adding $e_j.time$ to the following event in q_i, the kind of e_j is set to a dummy event and the event is not removed. If e_j is the last event in q_i then it is simply removed, together with any dummy events preceding it.

5.2 Support for periodic tasks

To support periodic tasks we introduce a new kind of RELTEQ events: a *period event*. Each period event e_i points to a task τ_i. The expiration of a period event e_i indicates the arrival of a periodic task τ_i upon which (i) the event time of the e_i is set to T_i and reinserted into the system queue using *InsertEvent()*, and (ii) the semaphore blocking τ_i is raised.

To support periodic tasks we have equipped each task with three additional variables: *TaskPeriod*, expressed in the number of ticks, *TaskPeriodSemaphore*, pointing to the semaphore guarding the release of the task, and *TaskPeriodEvent*, pointing to a RELTEQ period event. For efficiency reasons we have added these directly to the Task Control Block (TCB), which is the μC/OS-II structure storing the state information about a task. Our extensions could, however, reside in a separate structure pointing back to the original TCB.

A task τ_i is made periodic by calling *TaskMakePeriodic(τ_i, T_i)*, which

1. sets the *TaskPeriod* to T_i,
2. removes the *TaskPeriodEvent* from the system queue using *DeleteEvent()*, in case it was already inserted by a previous call to *TaskMakePeriodic()*, otherwise creates a new period event using *NewEvent(period, T_i, τ_i)*.
3. sets the event time of the *TaskPeriodEvent* to T_i and inserts it into the system queue.

6 Servers

In Section 5 we have introduced a system queue, which keeps track of the delay and period events for all tasks in the system. For handling periodic tasks assigned to servers we could reuse the system queue. However, this would mean that the tick handler would process the expiration of events local to inactive servers within the budget of the running server.

In order to limit the interference from inactive servers we would like to separate the events belonging to different servers. For this purpose we introduce additional RELTEQ queues for each server. We start this section by introducing additional primitives for manipulating queues, followed by describing how to use these in order to implement fixed-priority servers.

6.1 RELTEQ primitives for servers

We introduce the notion of a pool of queues, and define two pools: *active queues* and *inactive queues*. They are implemented as lists of RELTEQ queues. Conceptually, at every tick of the periodic timer the heads of all active queues are decremented. The inactive queues are left untouched.

To support servers we extend RELTEQ with the following methods:

ActivateQueue(q_i) Moves queue q_i from the inactive pool to the active pool.

DeactivateQueue(q_i) Moves queue q_i from the active pool to the inactive pool.

IncrementQueue(q_i) Increments the head event in queue q_i by 1. Time overflows are handled by setting the overflowing event to $2^n - 1$ and inserting a new dummy event at the head of the queue with time equal to the overflow (i.e. 1).

SyncQueueUntilEvent(q_i, q_j, e_k) Synchronizes queue q_i with queue q_j until event $e_k \in q_j$, by conceptually computing the absolute time of e_k, and then popping and handling all the events in q_i which have occurred during that time interval.

6.2 Limiting interference of inactive servers

To support servers, we add an additional *server queue* for *each* server σ_i, denoted by $\sigma_i.sq$, to keep track of the events local to the server, i.e. delays and periodic arrival of tasks $\tau_j \in \gamma(\sigma_i)$. At any time at most one server can be active; all other servers are inactive. The additional server queues make sure that the events local to inactive servers do not interfere with the currently active server.

When a server σ_i is switched in its server queue is activated by calling *ActivateQueue(σ_i.sq)*. In this new configuration the hardware timer drives two event queues:

1. the *system queue*, keeping track of system events, i.e. the replenishment of periodic servers,
2. the *server queue* of the *active* server, keeping track of the events local to a particular server, i.e. the delays and the arrival of periodic tasks belonging to the server.

When the active server is switched out (e.g. a higher priority server is resumed, or the active server gets depleted) then the active server queue is deactivated by calling *DeactivateQueue(σ_i.sq)*. As a result, the queue of the switched out server will be "paused", and the queue of the switched in server will be "resumed". The system queue is never deactivated.

To keep track of the time which has passed since the last server switch, we introduce a *stopwatch*. The stopwatch is basically a counter, which is incremented with every tick. In order to handle time overflows discussed in Section 4.1, we represent the stopwatch as a RELTEQ queue and use *IncrementQueue(stopwatch)* to increment it.

During the time when a server is inactive, several other servers may be switched in and out. Therefore, next to keeping track of time since the last server switch, for each server we also need to keep track of how long it was inactive, i.e the time since that particular server was switched out. Rather than storing a separate counter for each server, we multiplex the stopwatches for all servers onto the single stopwatch which we have already introduced, exploiting the RELTEQ approach. We do this by inserting a *stopwatch event*, denoted by $\sigma_i.se$, at the head of the stopwatch queue using *InsertEvent(stopwatch, σ_i.se)* whenever server σ_i is switched out. The event points to the server and its time is initially set to 0. The behavior of the tick handler with respect to the stopwatch remains unchanged: upon every tick the head event in the stopwatch queue is incremented using *IncrementQueue(stopwatch)*.

During runtime the stopwatch queue will contain one stopwatch event for every inactive server (the stopwatch event for the currently active server is removed when the server is switched in). The semantics of the stopwatch queue is defined as follows: the accumulated time from the head of the queue until (and including) a stopwatch event $\sigma_i.se$ represents the time the server σ_i was switched out.

When a server σ_i is switched in, its server queue is synchronized with the stopwatch using *SyncQueuesUntilEvent(σ_i.sq, stopwatch, σ_i.se)*, which handles all the events in $\sigma_i.sq$ which might have occurred during the time the server was switched out. It accumulates the time in the stopwatch queue until the stopwatch event $\sigma_i.se$ and handles all the events in $\sigma_i.sq$ which have expired during that time. Then $\sigma_i.se$ is removed from the stopwatch queue. When σ_i is switched out, $\sigma_i.se$ with time 0 is inserted at the head of the stopwatch queue.

When a server is switched in and no other server is switched out, then a dummy event e_i with time 0 is inserted at the head of the stopwatch using *InsertEvent(stopwatch, e_i)*. It makes sure that during the next tick the stopwatch will increment the new dummy event, rather than the stopwatch event corresponding to the previously switched out server.

Example of the stopwatch behavior The stopwatch queue is a great example of RELTEQ's strength. It provides an efficient and concise mechanism for keeping track of the inactive time for *all* servers. Figure 8 demonstrates the behavior of the stopwatch queue for an example system consisting of three servers A, B and C. It illustrates the state of the stopwatch queue at different moments during execution, before the currently running server is switched out and after the next server is switched in.

Initially, when server σ_i is created, a stopwatch event $\sigma_i.se$ with time 0 is inserted into the stopwatch queue. At time 0, when server A is switched in, no other server is switched out and hence a dummy event with time 0

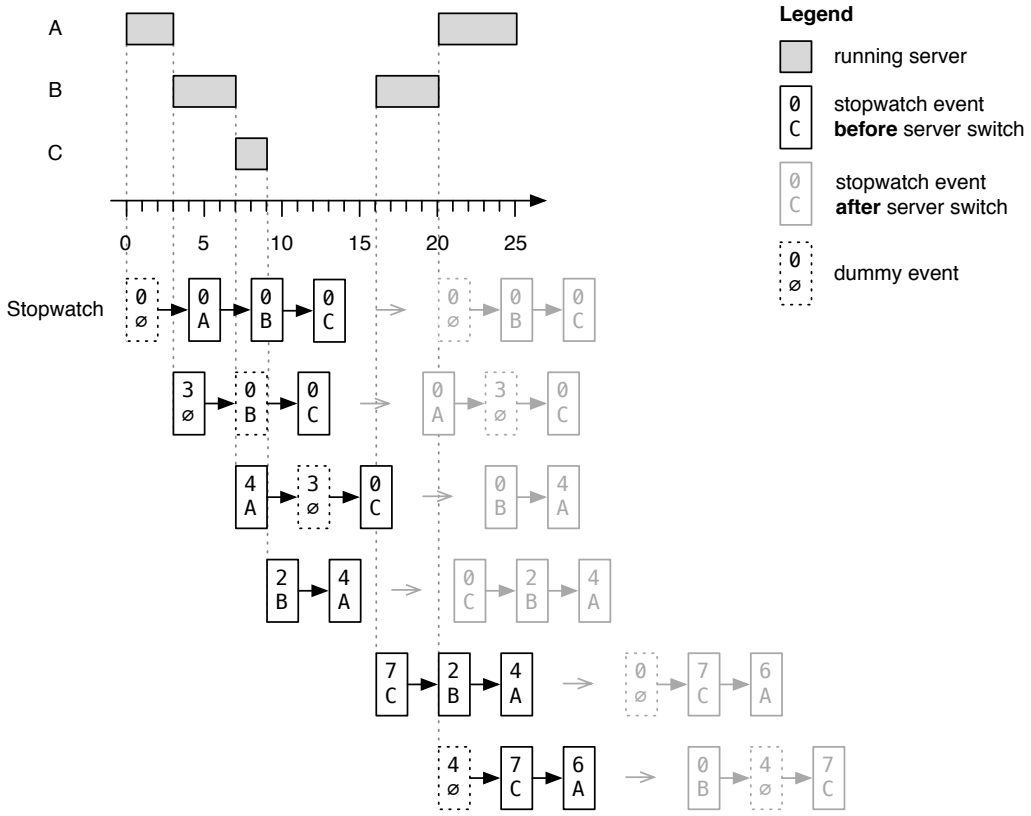


Figure 8: Example of the stopwatch queue.

is inserted. Also, since server A is switched in its stopwatch event is removed. While server A is running, the tick handler increments the head of the stopwatch queue, which happens to be the dummy event. At time 3, when server A is switched out, a new $\sigma_A.se$ with time 0 is inserted and $\sigma_B.se$ is removed.

At time 7 server C is switched in and $\sigma_C.se$ is deleted from the tail of the stopwatch queue. The trailing dummy event is also deleted, since it is no longer needed for the integrity of the queue.

At time 16, when server B is switched in after an idle period, no server is switched out, hence a dummy event with time 0 is inserted. Also, $\sigma_B.se$ is deleted from the stopwatch and its time is added to the following event in the queue, i.e. $\sigma_A.se$.

Deferrable server When the workload of a deferrable server σ_i is exhausted, i.e. there are no ready tasks in $\gamma(\sigma_i)$, then the server is switched out and its server queue $\sigma_i.sq$ is deactivated. Consequently, any periodic task which could wake up the server to consume any remaining budget cannot be noticed. One could alleviate this problem by keeping its server queue active when σ_i is switched out. This, however, would make the tick handler overhead linear in the number of deferrable servers, since a tick handler decrements the head events in all active queues (see Section 6.6).

Instead, in order to limit the interference of inactive deferrable servers, when a deferrable server σ_i is switched out and it has no workload pending (i.e. no tasks in $\gamma(\sigma_i)$ are ready), we deactivate the σ_i's server queue, change its state to waiting, and insert a *wakeup event*, denoted as $\sigma_i.we$, into the system queue. The wakeup event has its *data* pointing to σ_i and time equal to the arrival of the first event in $\sigma_i.sq$. When the wakeup event expires, the σ_i's state is set to the ready state. This way handling the events inside $\sigma_i.sq$ is deferred until σ_i is switched in.

When a deferrable server is switched in while it is in the waiting state, its wakeup event $\sigma_i.we$ is removed from the system queue.

Idling periodic server An idling periodic server is a special kind of a deferrable server containing an idle task (with lowest priority). The idle task is switched in if no higher priority task is ready, effectively idling away the

remaining capacity. In order to save memory needed for storing the task control block and the stack of the idle task, one idle task is shared between all idling periodic servers in the system.

6.3 Virtual timers

When the server budget is depleted an event must be triggered, to guarantee that a server does not exceed its budget. We present a general approach for handling budget depletion and introduce the notion of *virtual timers*, which are events relative to server's budget consumption.

We can implement virtual timers by adding a *virtual server queue* for *each* server, denoted by $\sigma_i.vq$. Similarly to the server queues introduced earlier, when a server is switched in, its virtual server queue is activated. The difference is that the virtual server queue is not synchronized with the stopwatch queue, since during the inactive period a server does not consume any of its budget. When a server is switched out, its virtual server queue is deactivated.

6.4 Switching servers

The methods for switching servers in and out are summarized in Figures 9 and 10.

SyncQueuesUntilEvent(σ_i.sq, stopwatch, σ_i.se);
ActivateQueue(σ_i.sq);
ActivateQueue(σ_i.vq);
if $\sigma_i.we \neq \emptyset$ **then**
  DeleteEvent(system, σ_i.we);
  $\sigma_i.we = \emptyset$*;*
end if

Figure 9: Pseudocode for *ServerSwitchIn(σ_i)*.

DeleteEvent(stopwatch, σ_i.se);
σ_i.se = NewEvent(stopwatch, 0, σ_i);
InsertEvent(stopwatch, $\sigma_i.se$);
DeactivateQueue(σ_i.sq);
DeactivateQueue(σ_i.vq);
if $\sigma_i.readyTasks = \emptyset$ **then**
  σ_i.we = NewEvent(wakeup, Head(σ_i.sq).time, σ_i);
  InsertEvent(system, σ_i.we);
end if

Figure 10: Pseudocode for *ServerSwitchOut(σ_i)*.

6.5 Server replenishment and depletion

We introduce two additional RELTEQ event kinds to support servers: *budget replenishment* and *budget depletion*. When a server σ_i is created, a replenishment event e_j is inserted into the *system queue*, with e_j*.data* pointing to σ_i and e_j*.time* equal to the server's replenishment period Π_i. When e_j expires, e_j*.time* is updated to Π_i and it is inserted into the system queue.

Upon replenishment, the server's depletion event e_j is inserted into its *virtual server queue*, with e_j*.data* pointing to σ_i and e_j*.time* equal to the server's capacity Θ_i. If the server was not depleted yet, then the old depletion event is removed from the virtual server queue using *DeleteEvent(σ_i.vq, e_j)*.

6.6 RELTEQ tick handler with support for servers

An example of the RELTEQ queues managed by the tick handler in the proposed RELTEQ extension with servers is summarized in Figure 11. Conceptually, every tick the stopwatch queue is incremented and the heads of the

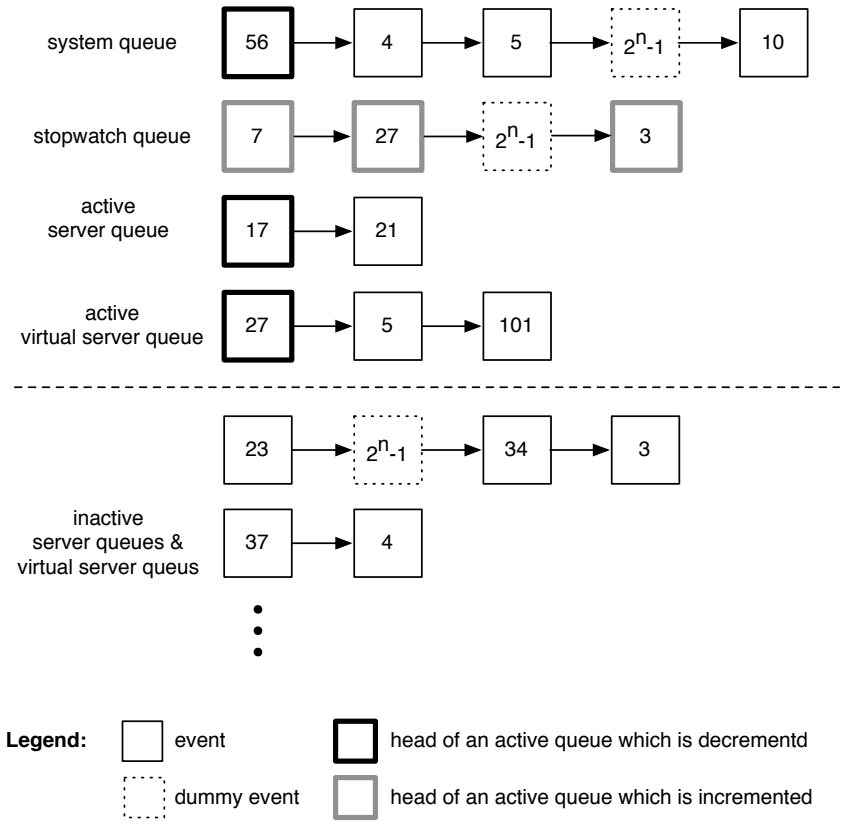


Figure 11: Example of the RELTEQ queues managed by the tick handler.

system queue, the active server queue and the active virtual server queue are decremented. If the head of any queue becomes 0, then their head event is popped and handled until the queue is exhausted or the head event has time larger than 0.

Actually, rather than decrementing the head of each active queue and checking whether it is 0, a *CurrentTime* counter is incremented and compared to the *Earliesttime*, which is set whenever the head of an active queue changes. If they are equal, then (i) the *CurrentTime* is subtracted from the heads of all the active queues, (ii) any head event with time 0 is popped and handled, (iii) *CurrentTime* is set to 0, and (iv) *Earliesttime* is set to the earliest time among the heads of all active queues[4].

The behavior of a RELTEQ tick handler supporting servers is summarized in Figure 12.

Note that at any moment in time there are at most four active queues, as shown in Figure 11.

Since we replace the standard μC/OS-II tick handler, it is indicated in Figure 1 as an interface provided by RELTEQ and required by μC/OS-II.

6.7 Summary

We have described a generic framework for supporting servers, which is tick based (Section 4.3) and limits the interference of inactive servers on system level (Section 6.2). The interference of inactive servers which are either ready or depleted was limited by means of a combination of inactive server queues and a stopwatch queue. Deactivating server queues of waiting servers was made possible by inserting a wakeup event into the system queue, in order to wake up the server upon the arrival of a periodic task while the server is switched out.

[4]Note that the $time$ of any event will never become negative.

```
IncrementQueue(stopwatch);
CurrentTime := CurrentTime + 1;
if Earliesttime = CurrentTime then
  for all q ∈ activequeues do
    Head(q).time := Head(q).time − CurrentTime;
    while Head(q).time = 0 do
      HandleEvent(Head(q));
      PopEvent(q);
    end while
  end for
  CurrentTime := 0;
  Earliesttime := Earliest(activequeues);
  Schedule();
end if
```

Figure 12: Pseudocode for the RELTEQ tick handler supporting hierarchical scheduling.

7 Hierarchical scheduling

[19] identified four ingredients necessary for guaranteeing resource provisions: admission, monitoring, scheduling and enforcement. In this section we describe how our implementation of HSF addresses each of them.

7.1 Admission

We allow admission of new subsystems only during the integration, not during runtime. The admission testing requires analysis for hierarchical systems, which is outside the scope of this paper.

7.2 Monitoring

There are two reasons for monitoring the budget consumption of servers: (i) handle the budget depletion and (ii) allow the assigned tasks to track and adapt to the available budget.

In order to notice the moment when a server becomes depleted we have introduced a virtual *depletion event* for every server, which is inserted into its virtual server queue. When the depletion event expires, then (i) the server's capacity is set to 0, (ii) its state is set to depleted, and (iii) the scheduler is called.

In order to allow tasks $\tau_j \in \gamma(\sigma_i)$ tracking the server budget β_i we equipped each server with a *budget counter*. Upon every tick the budget counter of the currently active server is decremented by one. We also added the method:

ServerBudget(σ_i) which returns the current value of β_i, and can be called by any task.

Note that the depletion event discussed earlier will make sure that a depleted server is switched out before the counter becomes negative.

7.3 Scheduling

The μC/OS-II the scheduler does two things: (i) select the highest priority ready task, and (ii) in case it is different from the currently running one, do a context switch. Our hierarchical scheduler replaces the original *OS_SchedNew()* method, which is used by the μC/OS-II scheduler to select the highest priority ready task.

It first uses the *global scheduler HighestReadyServer()* to select the highest priority ready server, and then the server's *local scheduler HighestReadyTask()*, which selects the highest priority ready task belonging to that server. This approach allows to implement different global and local schedulers, and also different schedulers in each server. Our fixed-priority global scheduler is shown in Figure 13.

The $currentServer$ is a global variable referring to the currently active server. Initially $currentServer = \emptyset$, where $\emptyset$ refers to a null pointer.

The scheduler first determines the highest priority ready server. Then, if the server is different from the currently active server, a server switch is performed, composed of 3 steps:

```
highestServer := HighestReadyServer();
if highestServer ≠ currentServer then
  if currentServer ≠ ∅ then
    ServerSwitchOut(currentServer);
  else
    event := NewEvent(dummy, 0, ∅);
    InsertEvent(stopwatch, event);
  end if
  if highestServer ≠ ∅ then
    ServerSwitchIn(highestServer);
  end if
  currentServer := highestServer;
end if
if currentServer ≠ ∅ then
  return currentServer.HighestReadyTask();
else
  return idleTask;
end if
```

Figure 13: Pseudocode for the hierarchical scheduler.

1. If there is a currently active server, then it is switched out, using *ServerSwitchOut()* described in Section 6.4. Otherwise, a dummy event with time 0 is inserted at the head of the stopwatch queue.
2. If there is a ready server, then it is switched in, using *ServerSwitchIn()* described in Section 6.4.
3. The $currentServer$ is updated.

Finally the highest priority task in the currently active server is selected, using the current server's local scheduler *HighestReadyTask()*. If no server is active, then the idle task is returned.

Since we replace the standard μC/OS-II scheduler, it is indicted in Figure 1 as an interface provided by RELTEQ and required by μC/OS-II.

7.4 Enforcement

When a server becomes depleted during the execution of one of its tasks (i.e. if a depletion event expires), the task will be preempted and the server will be switched out. This is possible, since we assume preemptive and independent tasks.

8 Evalulation

In this section we evaluate the behavior, modularity, memory footprint and performance of the HSF extension for RELTEQ. We have implemented the proposed design within μC/OS-II.

8.1 Behavior

Figure 14 shows a trace of an example application consisting of two servers, each serving one task. We ran the setup within the OpenRISC simulator [17], with 1 tick set to 1ms[5]. The task execution is shown on top, with the server capacities illustrated underneath. The application was traced and visualized using the Grasp toolset [11].

In this particular example Task1 is assigned to the Deferrable Server, and Task2 is assigned to the Periodic Server. Both tasks have a period of 30ms and execution time of 5ms. Task1 has an offset of 5ms. Both servers have a replenishment period of 25ms and capacity of 10ms. The figure demonstrates, how the idling periodic server idles away its capacity when there is no workload pending (time interval 10-15ms), while the deferrable server

[5]The complete simulation setup and the source code of our HSF realization are available at `http://www.win.tue.nl/~mholende/ucos/`.

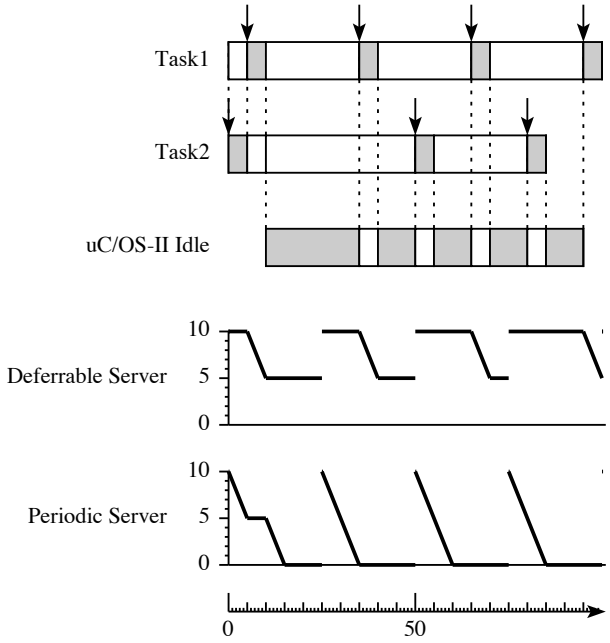


Figure 14: Example trace of an application with one deferrable and one periodic idling server.

preserves its capacity until Task1 arrives (time interval 25-35ms). The figure also illustrates how the handling of period events (indicated by the arrows) of Task2 is postponed until the Periodic Server is switched in (at time 50ms).

In summary, the trace shown in Figure 14 exhibits correct behavior of periodic-idling and deferrable servers.

8.2 Modularity

The design of RELTEQ and the HSF extension is modular, allowing to enable or disable the support for HSF and different server types during compilation with a single compiler directive for each extension.

The complete RELTEQ implementation including the HSF extension is 1610 lines of code (excluding comments and blank lines), compared to 8330 lines of the original μC/OS-II. 105 lines of code were inserted into the original μC/OS-II code, out of which 60 were conditional compilation directives allowing to easily enable and disable our extensions. No original code was deleted or modified. Note that the RELTEQ+HSF code can replace the existing timing mechanisms in μC/OS-II, and that it provides a framework for easy implementation of other scheduler and servers types.

The 105 lines of code represent the effort required to port RELTEQ+HSF to another operating system. Such porting would require (i) redirecting the tick handler to the RELTEQ handler, (ii) redirecting the method responsible for selecting the the highest priority task to the HSF scheduler, and (iii) identifying when tasks become ready or blocked.

The code memory footprint of RELTEQ+HSF is 8KB, compared to 32KB of the original μC/OS-II. The additional data memory foot print for an application consisting of 6 servers with 6 tasks each is 5KB, compared to 47KB for an application consisting of 36 tasks (with a stack of 128B each) in the original μC/OS-II.

8.3 Performance analysis

In this section we evaluate the system overheads of our extensions, in particular the overheads of the scheduler and the tick handler.

Scheduler The scheduling overhead for HSF is linear in the number of servers, since they need to be traversed in order to determine the highest priority ready server.

The overhead of switching in a server depends on the number of events inside the stopwatch queue and the server queue of the switched in server, which needs to be synchronized with the stopwatch. The stopwatch queue contains one stopwatch event per server and at most $k = \frac{t_l}{2^n-1} + |\Sigma|$ dummy events, where t_l is the longest time interval that a server can be switched out, and $2^n - 1$ is the largest relative time which can be represented with n bits. The number of servers $|\Sigma|$ represents the worst-case number of dummy events in the stopwatch queue which are inserted when a server is switched in without another server being switched out. Since the only local events are a task delay and the arrival of a periodic task, the number of events inside the server queue which need to be handled is bounded by the number of tasks assigned to the switched in server.

Selecting the highest priority task within a server is linear in the number of tasks assigned to a server[6].

Therefore, the complexity of the scheduler is $O(|\Sigma| + m + k)$, where $|\Sigma|$ is the number of servers, m is the maximum number of tasks assigned to a server, and k is the maximum number of dummy events inside the stopwatch queue.

Tick handler The tick handler synchronizes all active queues with the current time, and (in case an event was handled) calls the scheduler. There is one system queue and two queues for the currently switched in server. The system queue contains only replenishment and wakeup events, therefore its size is proportional to the number of servers. The size of the active server queue and virtual server queue is linear in the number of tasks assigned to the server. Similarly, since the active virtual server queue contains one depletion event and at most one virtual timer for each task, its size is linear in the number of tasks assigned to the server. The tick handler complexity is therefore $O(|\Sigma| + m)$.

Alternative approach: no wakeup events In Section 6.2 we introduced wakeup events in order to limit the interference due to inactive servers. In order to validate this approach we have also implemented a variant of the HSF scheduler which avoids using wakeup events and instead, whenever a deferrable server σ_i is switched out, it keeps the server queue $\sigma_i.sq$ active. Consequently, the scheduler does not need to synchronize the server queue when switching in a server. However, this overhead is shifted to the tick handler, which needs to handle the expired events in all the server queues from inactive deferrable servers.

Figures 15.a to 15.e compare the two variants. We have varied the number of deferrable servers and the number of tasks assigned to each server (all servers had the same number of tasks). The server replenishment periods and the task periods were all set to the same value (100ms), to exhibit the maximum overhead by having all tasks arrive at the same time. Each task had a computation time of 1ms and each server had a capacity of 7ms. We have run the setup within the cycle-accurate hardware simulator for the OpenRISC 1000 architecture [17]. We have set the processor clock to 345MHz and the tick to 1KHz, which is inline with the platform used by our industrial partner.

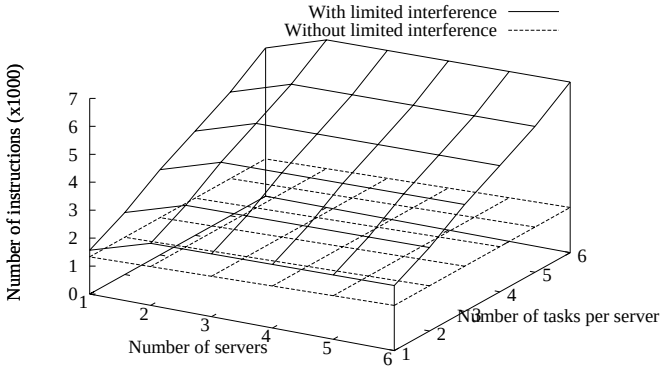
Figures 15.a and 15.b show the maximum measured overheads of the scheduler and the tick handler, while Figures 15.c and 15.d show the total overhead of the scheduler and the tick handler in terms of processor utilization. The figures demonstrate that wakeup events reduce the tick overhead, at the cost of increasing the scheduler overhead, by effectively shifting the overhead of handling server's local events from the tick handler to the scheduler. Since the scheduler overhead is accounted to the server which is switched in, as the number of servers and tasks per server increase so does the advantage of the limited interference approach. Figure 15.e combines Figures 15.c and 15.d and verifies that the additional overhead due to the wakeup events in the limited interference approach is negligible.

Figure 15.f compares the system overheads of our HSF extension to the standard μC/OS-II implementation. As the standard μC/OS-II does not implement hierarchical scheduling, we have simulated a flat system containing the same number of tasks with the same parameters as in Figures 15.a and 15.b. The results demonstrate the temporal isolation and efficiency of our HSF implementation. While the standard μC/OS-II scans through all tasks on each tick to see if any task delay has expired, in the HSF extension the tick handler needs to consider only head event in the server queue of the currently running server.

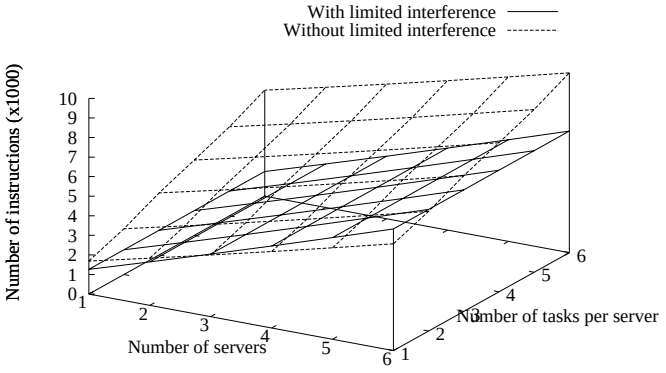
Figure 16 compares the best-case and worst-case measured overheads of the scheduler and tick handler between μC/OS-II with our extensions, compared to the standard μC/OS-II, for which we have simulated a flat system containing the same number of tasks with the same parameters as for the μC/OS-II+HSF case.

The figure shows that both scheduler and tick handler suffer larger execution time jitter under μC/OS-II+HSF, than the standard μC/OS-II. In the best case the μC/OS-II+HSF tick handler needs to decrement only the head of

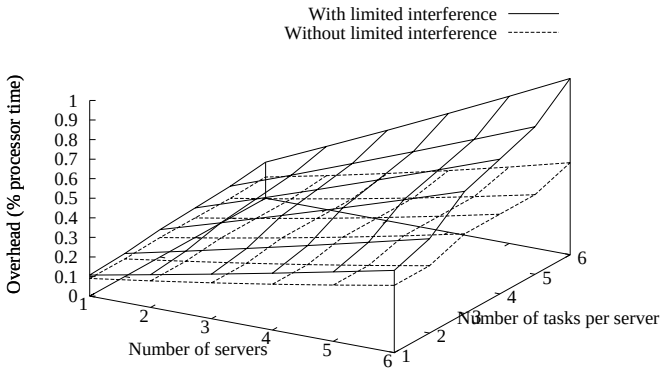
[6]Note that selecting the highest priority server or task can be done in constant time, by selecting the first server/task from a ready queue ordered by priority, at the cost of a linear complexity of inserting the server/task into the ready queue whenever a server/task becomes ready.



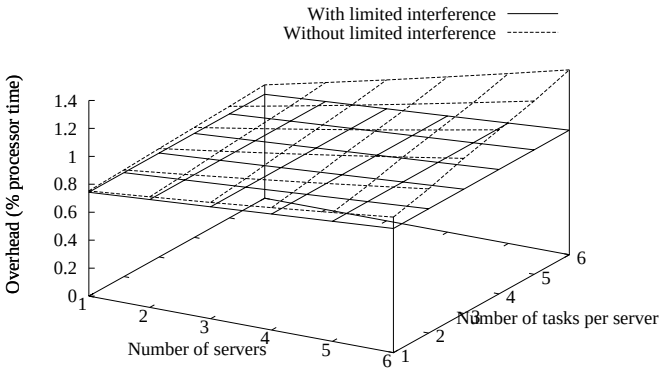
(a)



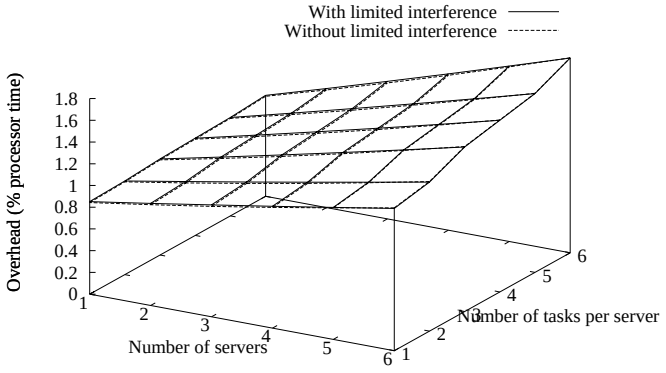
(b)



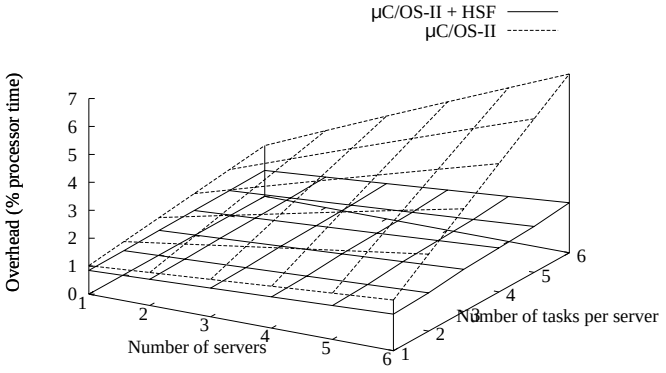
(c)



(d)



(e)



(f)

Figure 15: (a) maximum overhead of the scheduler, (b) maximum overhead of the tick handler, (c) total overhead of the scheduler, (d) total overhead of the tick handler, (e) and (f) cumulative overhead of the tick handler and the scheduler.

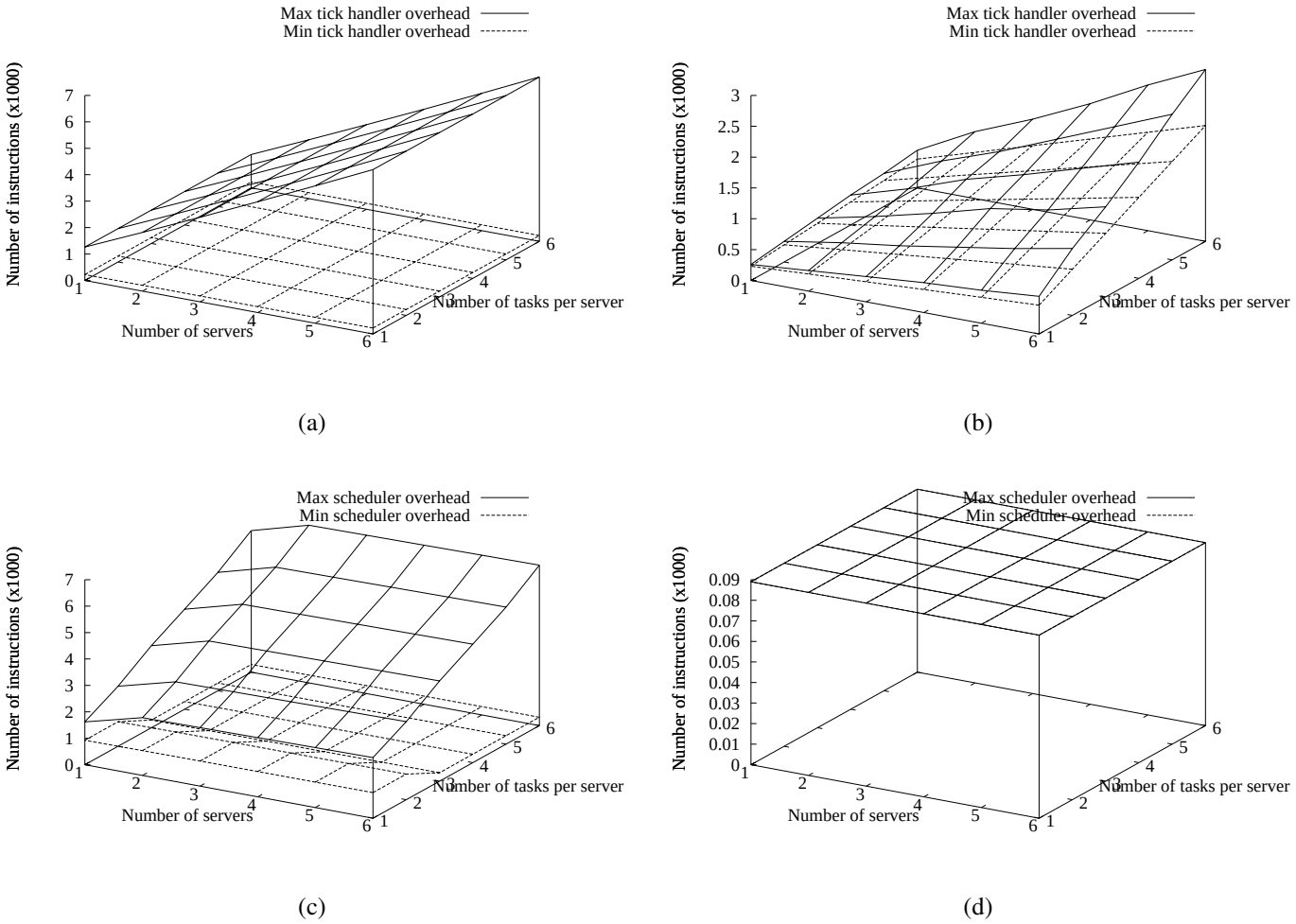


Figure 16: (a) tick handler overhead in μC/OS-II+HSF, (b) tick handler overhead in μC/OS-II, (c) scheduler overhead in μC/OS-II+HSF, (d) scheduler overhead in μC/OS-II.

the system queue, while in μC/OS-II the tick hander traverses all the tasks in the system and for each one it checks whether its timer has expired.

In a system with small utilization of individual tasks and servers (as was the case in our simulations), most local events will arrive while the server is switched out. Since handling local events is deferred until the server is switched in and its server queue synchronized with the stopwatch queue, it explains why the worst-case tick handler overhead is increasing with the number of servers and the worst-case scheduler overhead is increasing with the number of tasks per server.

While in μC/OS-II+HSF we use linked lists to represent the ready queues for each server, μC/OS-II represents the ready queue as a two dimensional bit-map table allowing its scheduler to identify ready tasks and change task's ready state in constant time.

9 Conclusions

In this paper we have presented an efficient, modular and extendible design for enhancing a real-time operating system with periodic tasks, polling, idling periodic and deferrable servers, and two-level fixed-priority HSF. It relies on Relative Timed Event Queues (RELTEQ), a timed event management component targeted at embedded operating systems. Our design supports global and virtual timers, provides temporal isolation between subsystems and limits the interference of inactive servers on the active server, by means of wakeup events and a combination

of inactive server queues with a stopwatch. We have implemented it and evaluated within the μC/OS-II real-time operating system used by our industrial and academic partners. Our approach is modular, exhibits low performance overhead and minimizes the necessary modifications of the underlying operating system.

In the future we would like to apply our RELTEQ based HSF approach in other real-time operating systems, such as VxWorks.

Acknowledgements

We thank Martijn van den Heuvel for the insightful discussions and extensive feedback on earlier versions of this document.

References

[1] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril, "Towards hierarchical scheduling on top of VxWorks," in *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, July 2008, pp. 63–72.

[2] G. Buttazzo and P. Gai, "Efficient EDF implementation for small embedded systems," in *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2006.

[3] A. Carlini and G. C. Buttazzo, "An efficient time representation for real-time embedded systems," in *ACM Symposium on Applied Computing*, 2003, pp. 705–712.

[4] R. I. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in *Real-Time Systems Symposium (RTSS)*, 2005, pp. 389–398.

[5] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: an operating system architecture for application-level resource management," in *Symposium on Operating Systems Principles (SOSP)*, 1995, pp. 251–266.

[6] A. Eswaran, A. Rowe, and R. Rajkumar, "Nano-RK: An energy-aware resource-centric RTOS for sensor networks," in *Real-Time Systems Symposium (RTSS)*, 2005, pp. 256–265.

[7] Evidence. ERIKA Enterprise: Open Source RTOS for single- and multi-core applications. [Online]. Available: http://www.evidence.eu.com

[8] D. Faggioli, M. Trimarchi, F. Checconi, and C. Scordino, "An EDF scheduling class for the linux kernel," in *Real-Time Linux Workshop*, 2009.

[9] M. Holenderski, R. J. Bril, and J. J. Lukkien, "Using fixed priority scheduling with deferred preemption to exploit fluctuating network bandwidth," in *Work in Progress session of the Euromicro Conference on Real-Time Systems (ECRTS)*, 2008.

[10] M. Holenderski, W. Cools, R. J. Bril, and J. J. Lukkien, "Multiplexing real-time timed events," in *Work in Progress session of the International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2009.

[11] M. Holenderski, M. M. H. P. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Grasp: Tracing, visualizing and measuring the behavior of real-time systems," in *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, July 2010.

[12] D. Kim, Y.-H. Lee, and M. Younis, "SPIRIT-μKernel for strongly partitioned real-time systems," in *International Conference on Real-Time Systems and Applications (RTCSA)*, 2000, pp. 73–80.

[13] J. J. Labrosse, *MicroC/OS-II: The Real-Time Kernel*. CMP Books, 1998.

[14] J. P. Lehoczky, L. Sha, and J. K. Strosnider, "Enhanced aperiodic responsiveness in hard real-time environments," in *Real-Time Systems Symposium (RTSS)*, 1987, pp. 261–270.

[15] C. Mercer, R. Rajkumar, and J. Zelenka, "Temporal protection in real-time operating systems," in *IEEE Workshop on Real-Time Operating Systems and Software*, 1994, pp. 79–83.

[16] S. Oikawa and R. Rajkumar, "Portable RK: a portable resource kernel for guaranteed and enforced timing behavior," in *Real-Time Technology and Applications Symposium (RTAS)*, 1999, pp. 111–120.

[17] OpenCores. Openrisc 1000: Architectural simulator. [Online]. Available: http://opencores.org/openrisc, or1ksim

[18] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "Aquosa—adaptive quality of service architecture," *Software – Practice and Experience*, vol. 39, no. 1, pp. 1–31, 2009.

[19] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource kernels: A resource-centric approach to real-time and multimedia systems," in *Conference on Multimedia Computing and Networking (MMCN)*, 1998, pp. 150–164.

[20] S. Saewong and R. Rajkumar, "Hierarchical reservation support in resource kernels," in *Real Time Systems Symposium (RTSS)*, 2001.

[21] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Real-Time Systems Symposium (RTSS)*, 2003, pp. 2–13.

[22] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Transactions on Computers*, vol. 44, no. 1, pp. 73–91, 1995.